

SMART HEALTH MONITORING SYSTEM

Submitted in the partial fulfillment for the requirements

For the award of the degree of

Bachelor of Engineering

in

Electronics and Telecommunication Engineering

By

Samarth Sawant (B-836)

Sanskar Patil (B-840)

Harsh Singh (B-829)

Rishikesh Patil (B-837)

Under the Supervision of

Prof. Kavita Rathi

MANJARA CHARITABLE TRUST
RAJIV GANDHI INSTITUTE OF TECHNOLOGY, MUMBAI
(Permanently Affiliated to University of Mumbai)

(Department of Electronics and Telecommunication Engineering)

(University of Mumbai)

Academic Year 2024-25

Certificate

This is to certify that the project entitled **SMART HEALTH MONITORING SYSTEM** is a bona fide work of **Samarth Sawant (Roll No. B-836)**, **Sanskar Patil (Roll No. B-840)**, **Harsh Singh (Roll No. B-829)** and **Rishikesh Patil (Roll No. B-837)** under the supervision of **Prof. Kavita Rathi**, submitted to the University of Mumbai in partial fulfillment of the requirement for the award of the degree of **Bachelor of Engineering in Electronics and Telecommunication Engineering**.

Prof. Kavita Rathi
Supervisor

Dr. Sanjay D. Deshmukh
Head of Department

Dr. Sanjay U. Bokade
Principal

REPORT APPROVAL FOR B.E.

This project report entitled “**SMART HEALTH MONITORING SYSTEM**” by **Samarth Sawant, Sanskar Patil, Harsh Singh and Rishikesh Patil** is approved for the degree of Bachelor of Engineering in Electronics and Telecommunication Engineering from University of Mumbai, in academic year 2024-25.

Examiners:

1. Internal: _____

1. External: _____

Date:

Place: **Mumbai**

Declaration

We, the undersigned, declare that the project entitled **SMART HEALTH MONITORING SYSTEM** is a bona fide work carried out by us for the partial fulfillment of the requirements for the degree of Bachelor of Engineering in Electronics and Telecommunication Engineering at Rajiv Gandhi Institute of Technology, Mumbai. The work presented in this report is original and has not been submitted earlier to any other university or institution for the award of any degree or diploma.

Samarth Sawant

Roll No. B-836

Sanskar Patil

Roll No. B-840

Rishikesh Patil

Roll No. B-837

Harsh Singh

Roll No. B-829

Prof. Kavita Rathi

Project Guide

Abstract

This document presents a Smart Health Monitoring System that uses intelligent technology to support patient care and maintenance in the Intensive Care Unit (ICU). Traditional ICU care systems often rely on manual assessments, which can be slow to identify significant changes in a patient's health. The integration of IoT sensors, data analytics, and communication technologies helps automate the monitoring process, providing real-time, continuous, and personalized care.

This system allows for:

- Real-time monitoring of vital parameters like heart rate, SpO₂, and temperature.
- Automated alerts sent to medical staff via mobile notifications when patient vitals exceed safe thresholds.
- Secure and scalable data storage using a cloud-based platform.
- Predictive analytics to identify potential health risks and enable timely intervention.

The aim of this project is to improve the efficiency of monitoring systems in ICUs, reduce the burden on healthcare professionals, and ensure that patients receive timely intervention when required.

Keywords: Smart Health Monitoring, IoT in Healthcare, ICU Automation, Vital Parameter Monitoring, Real-Time Alert System, Cloud Health Data, Predictive Health Analytics

Contents

Abstract	4
List of Figures	8
List of Tables	9
1 Introduction	10
2 Problem Statement and Objectives	11
2.1 Problem Statement	11
2.2 Objectives	12
3 Literature Survey	13
4 Methodology	14
4.1 System Design and Architecture	14
4.1.1 Patient-side System Architecture	14
4.1.2 Integration with the Backend	17
4.2 Data Transmission and Cloud Integration	18
4.3 Dashboard Development	18
4.4 Automated Alerts and Notifications	19
4.5 Predictive Analytics and Machine Learning (Optional)	19
4.6 Testing and Validation	19
4.7 Security and Data Privacy Implementation	19
5 Requirements	20
5.1 Hardware Requirements	20
5.2 Software Requirements	21
5.3 Network and Communication Requirements	21
5.4 Security and Privacy Requirements	22
6 Components and Technologies	23
6.1 Frontend Technologies	23

6.1.1	Frameworks Used	23
6.1.2	Additional Libraries	24
6.2	Backend Technologies	24
6.2.1	Frameworks Used	24
6.2.2	Folder Structure	25
6.2.3	Middleware	25
6.2.4	Additional Libraries Used	26
6.2.5	Real-Time Data Streaming with Socket.IO	26
6.2.6	Containerization with Docker	28
6.2.7	Redis: In-memory Data Store	30
6.2.8	MongoDB: Database for Patient Data Storage	33
6.2.9	Arduino IDE: Programming the Sensors	35
7	Implementation Details	37
7.1	Code Implementation	37
7.1.1	Medical History Page	37
7.1.2	Alert System	38
7.2	Sensor Simulation	39
7.3	Live Sensor Readings	40
7.4	PDF Generation	42
7.5	Redis Implementation and Server Load Balancing	44
7.5.1	How Redis is Implemented	44
7.5.2	How Redis Helps the Smart Health Monitoring System	44
7.6	Docker Implementation and Containerization	45
7.6.1	How Docker is Implemented	45
7.6.2	How Docker Helps the Smart Health Monitoring System	45
7.7	MongoDB Implementation and Data Management	46
7.7.1	How MongoDB is Implemented	46
7.7.2	How MongoDB Helps the Smart Health Monitoring System	46
7.8	Arduino Code Implementation and Real Sensor Data Acquisition	46
7.8.1	How Arduino Code is Implemented	47
7.8.2	How Arduino Code Helps the Smart Health Monitoring System	47
7.9	ESP8266 Code Implementation and Wireless Transmission of Real Sensor Data	48
7.9.1	How ESP8266 Code is Implemented	48
7.9.2	How ESP8266 Code Helps the Smart Health Monitoring System	48
8	Results and Discussion	49
8.1	Screenshots/Visuals	49
8.2	Testing and Validation	50

8.2.1	Patient Location Tracking	50
8.2.2	Medical History Management	51
8.2.3	Alert System	52
8.3	Challenges and Solutions	53
8.3.1	Google Maps Integration	53
8.3.2	Managing Simulation Data	53
8.3.3	Managing Cached Live Sensor Data	53
8.3.4	Managing Server Load with Real-Time Streaming	54
8.3.5	Efficient Storage of Live Sensor Data	55
8.3.6	User Interface and Responsiveness	56
9	Future Work and Enhancements	57
9.1	Potential Improvements	57
9.1.1	Integration of Real Sensors	57
9.1.2	Advanced Alert System with AI Integration	57
9.1.3	Cloud-Based Data Storage and Remote Access	58
9.1.4	Mobile App Integration	58
9.1.5	Scalability for Large-Scale Implementation	58
9.1.6	IoT and Wearable Device Connectivity	59
10	Conclusion	60

List of Figures

4.1	Smart Health Monitoring System Architecture	14
4.2	Smart Health Monitoring System Flow Chart	15
6.1	Smart Health Monitoring Backend Docker Container	30
8.1	Patient-Side Dashboard displaying Health Metrics and Patient Status	49
8.2	GPS Tracking Page showing Patient's Live Location on Google Maps	51
8.3	Medical History Page displaying Historical Health Data and Patient's Health Profile	52
8.4	Alert System Page showing Critical Health Alerts such as Abnormal Tempera- ture or Heart Rate	53

List of Tables

Chapter 1

Introduction

The intensive care unit (ICU) is a critical hospital environment where critically ill and lifethreatening patients are cared for. Patient care in the ICU is complex and requires constant monitoring of vital parameters such as heart rate, blood pressure, blood oxygen level (SpO2), and body temperature. Traditionally, patient care in the ICU has relied heavily on bedside equipment and physician interventions. However, these methods can sometimes delay detection of emergencies or subtle changes in a patient's health, affecting patient outcomes. With the rise of the Internet of Things (IoT) and cloudbased technology, the healthcare industry has begun exploring new ways to increase the efficiency and reliability of patient care. Intelligent Patient Health Monitoring Systems Leverage these advances to address current limitations. The system is designed to collect realtime data from sensors attached to the patient, process it using cloud computing, and automatically alert doctors to detect defects during the exam. This approach ensures timely medical intervention and reduces the risk of death. Remote access. This continuous monitoring can also help reduce the workload of ICU staff, allowing them to focus on other important tasks while being alerted to crisis situations. The system's scalability makes it suitable for a variety of ICU settings, and multiple sensors can be added or integrated with hospital data for seamless health management. We're making ICU care smarter and more efficient. Smart healthcare is not just a technological innovation, but also a step towards making healthcare more efficient. The system works to save lives and improve the quality of care by increasing the reliability, speed and accessibility of important patient information.

Chapter 2

Problem Statement and Objectives

2.1 Problem Statement

- Intensive care units (ICUs) play a vital role in saving lives, but current care systems have limitations that can impact patient outcomes. Routine critical care monitoring involves monitoring vital signs and relying on medical staff to detect abnormalities and take action quickly. When nurses and doctors are stressed or when monitoring equipment doesn't sound the alarm immediately, important events can be missed and treatment can be delayed. Additionally, the lack of remote access to patient information can be problematic because decisions can't be made immediately. Risk of delay
- Overcrowding of healthcare staff: As the number of patients increases, it becomes difficult for doctors to regularly attend to all their patients, which can lead to human error. Warning to delay medical intervention in critical situations. The system cannot use diagnostic tools to detect early signs of damage, reducing the risk of necessary care
- Manual Monitoring Limitations: Traditional ICU monitoring depends heavily on manual observation, increasing the risk of delays in detecting emergencies
- Overburdened Healthcare Staff: With increasing patient loads, healthcare personnel may struggle to monitor every patient continuously, leading to human errors.
- Lack of Real-Time Alerts: Conventional systems may not provide instant alerts, resulting in delayed medical interventions during critical situations
- No Remote Access: Patient data is often accessible only at the bedside, limiting doctors' ability to monitor patients remotely.

2.2 Objectives

- **Real-Time Monitoring:** Capture vital parameters such as heart rate, SpO2, blood pressure, and body temperature continuously using IoT sensors.
- **Automated Alerts:** Develop a notification system to instantly alert medical staff of abnormalities via mobile or web platforms
- **Remote Access:** Enable healthcare professionals to monitor patient data remotely for timely decision-making
- **Predictive Analytics:** Use data analytics to identify health patterns and predict potential risks for proactive care.
- **Scalable System:** Design a flexible system that allows integration with additional sensors or hospital systems based on requirements
- **Data Security:** Ensure patient privacy and data security through encryption and compliance with healthcare regulations

Chapter 3

Literature Survey

Recent advancements at the intersection of medicine and technology have enabled the development of more intelligent, real-time healthcare monitoring systems. Several research works have contributed concepts and methods that influenced the design and implementation of our Smart Health Monitoring System. The following survey highlights major outcomes from key studies and how they informed our project development:

1. **Baig and Gholamhosseini (2013)** presented a comprehensive study on smart health monitoring systems, focusing on design and integration strategies. Their emphasis on real-time vital monitoring, data acquisition modules, and system modeling directly influenced the modular structure of our Smart Health Monitoring system, including multi-sensor simulation and continuous data acquisition techniques [1].
2. **Kim et al. (2014)** explored interference challenges between ZigBee-based Wireless Body Area Networks (WBANs) and WiFi systems in health telemonitoring. Based on their findings, our implementation considered reliable serial communication for sensor data transmission, and an optional wireless (Wi-Fi/ESP8266) module, ensuring minimized interference and consistent data delivery [2].
3. **Riazulislam et al. (2015)** conducted an extensive survey on the application of IoT in healthcare. Their insights into remote patient monitoring, wearable device data acquisition, and real-time analytics inspired our use of simulated IoT-based sensor networks, cloud data storage concepts, and predictive alert generation in the Smart Health Monitoring environment [3].
4. **Al Alkeem et al. (2017)** proposed a secure healthcare system integrating Cloud of Things (CoT) technologies. Inspired by their work, our backend architecture ensures secure, scalable data handling through cloud platforms (MongoDB, Redis) and emphasizes secure real-time health data transmission from the device to the healthcare dashboard [4].

Chapter 4

Methodology

4.1 System Design and Architecture

4.1.1 Patient-side System Architecture

The **Smart Health Monitoring** system is designed to monitor a patient's health status and provide real-time updates to healthcare providers through a web-based frontend. The system architecture consists of multiple components working in unison to collect, simulate, display, and manage the health data of the patient.

Block Diagram

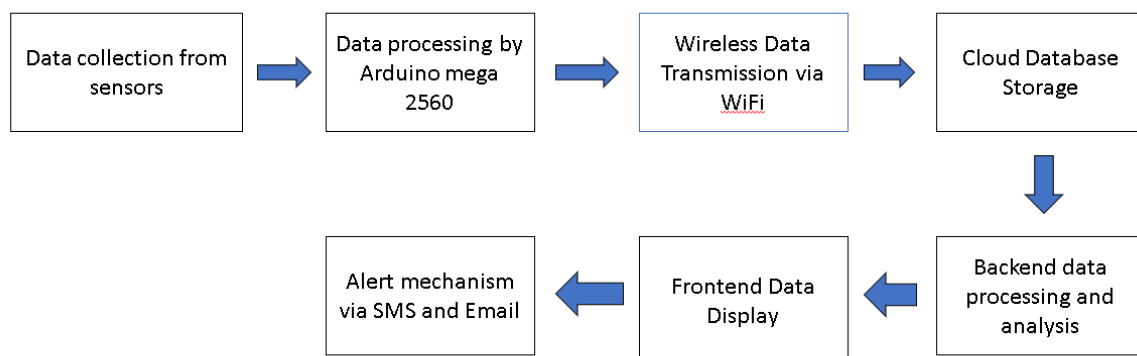


Figure 4.1: Smart Health Monitoring System Architecture

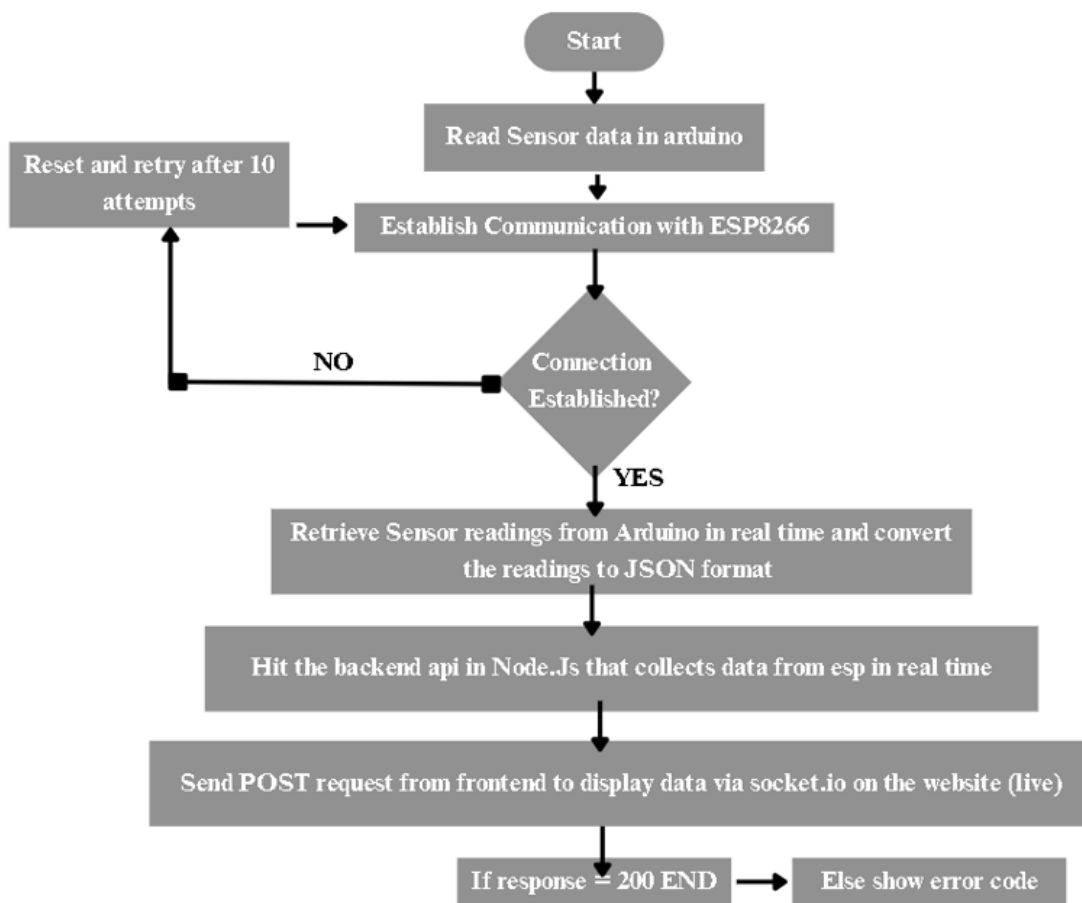


Figure 4.2: Smart Health Monitoring System Flow Chart

Flow Chart

Components Used

- **Arduino Mega 2560:** Acts as the central processing unit in the system. It simulates various sensors like heart rate, body temperature, and other health metrics. It communicates with the sensors (real or simulated), collects data, and transmits it for processing or visualization.
- **16x2 LCD:** This display is used to show critical health information to the healthcare provider or patient on-site, such as heart rate, body temperature, and patient status. In the simulation, this is simulated using the frontend, as the LCD is replaced by dynamic displays (e.g., React UI).
- **Simulated Sensors:**
 - **Air Quality Sensor (e.g., MQ-135):** Measures the quality of air surrounding the patient, detecting pollutants such as CO₂, alcohol, ammonia, and other gases. The sensor's output is simulated with varying air quality values to represent environmental conditions.

- **Toxic Gas Sensor (e.g., MQ-7, MQ-9):** Detects the presence of toxic gases such as carbon monoxide (CO) and other harmful substances. In the simulation, the values are simulated to monitor the exposure to harmful gases.
 - **ECG Sensor (e.g., AD8232):** Monitors the electrical activity of the heart by measuring the ECG signal. In the simulation, the ECG values are generated to represent the patient's heart rhythm and condition.
 - **SpO2 Sensor (e.g., MAX30100):** Measures the oxygen saturation in the blood. The simulation mimics the real sensor data, with values ranging from 95-100
 - **Body Movement Sensor (e.g., MPU6050 Accelerometer):** Tracks the movement and orientation of the patient. The sensor data is simulated in the frontend to detect any significant body movements such as walking or changes in posture.
 - **Pressure Sensor (e.g., BMP180):** Measures the atmospheric pressure and temperature around the patient. In the simulation, the values are generated to mimic real-world atmospheric pressure data.
 - **LM35 Temperature Sensor:** Monitors the patient's body temperature. In the simulation, the temperature value is randomly generated to simulate real-time body temperature.
 - **Humidity Sensor (e.g., DHT11/DHT22):** Measures the relative humidity of the environment. The sensor value is simulated with varying levels of humidity, which may impact the patient's comfort and health.
 - **Alarm System:** The alarm system is used to notify healthcare providers when a sensor reading exceeds predefined thresholds, such as abnormal temperature, heart rate, or SpO2 levels. In the simulation, the alarm is triggered when critical values are detected, alerting the staff.
- **GPS Tracking:** Simulated in the frontend, this system allows for real-time tracking of the patient's movement. It uses the patient's location data (latitude and longitude) to update a Google Map and show the live location on the user interface.

Working

- **Sensors:** These components continuously collect data related to the patient's health. The Arduino Mega processes the sensor outputs and sends them to the cloud or the frontend for visualization. The sensors include the air quality sensor, toxic gas sensor, ECG sensor, SpO2 sensor, body movement sensor, pressure sensor, LM35 temperature sensor, and humidity sensor, all of which provide critical health and environmental data.
- **Simulation:** In the absence of real hardware, the sensors' data (e.g., temperature, heart rate, body movement, air quality) is simulated in the frontend. The simulated data mimics

real sensor outputs, allowing healthcare providers to monitor the patient's condition in real time. For example, temperature data is randomly generated, heart rate is simulated, and movement data is generated to track the patient's physical activity.

- **Frontend (Web Interface):** The React application provides a user-friendly interface to display the patient's health data and live location. It consists of several pages, such as:
 - **Patient Location:** Displays the current GPS location of the patient on a map using real-time latitude and longitude coordinates.
 - **Medical History:** Tracks historical medical data of the patient, including past health records, vitals, and sensor data.
 - **Alerts:** Shows any critical health alerts (e.g., abnormal vitals) triggered when sensor data exceeds certain thresholds. For example, an abnormal heart rate or SpO2 level will trigger an alert.
 - **Health Monitoring:** Real-time health parameters like temperature, heart rate, ECG, and SpO2 are displayed and updated periodically. This allows healthcare providers to continuously monitor the patient's vital signs.

4.1.2 Integration with the Backend

In this Smart Health Monitoring system, the data from the sensors or simulations is transferred from the **Arduino Mega** to the **cloud** and then displayed on the **frontend** (web interface). This integration can be described in several stages:

Data Collection from Sensors

The Arduino Mega collects data from the simulated sensors (e.g., heart rate, body temperature, GPS location). This data is processed and stored temporarily on the Arduino Mega. In a real-world scenario, this data would be transmitted to the cloud via an IoT platform or network protocol.

Data Transfer to Cloud

In a full implementation, the Arduino could be connected to the cloud via Wi-Fi or GSM modules, sending the data to a cloud server where it is stored in a database. For the purposes of this simulation, the data is transferred to the cloud using simulated API requests. This API can serve as an intermediary between the Arduino and the web interface.

Backend Technologies: This could involve the use of cloud platforms such as **AWS**, **Google Cloud**, or **Microsoft Azure** for storing patient data securely.

Data Retrieval by Frontend

The frontend (React web interface) fetches data from the cloud (or simulated APIs) in real-time. Data such as the patient's **heart rate**, **body temperature**, and **location** are displayed dynamically on the web pages. The **Google Map** component receives the patient's GPS location in real-time, updating the map to reflect the patient's current position.

Alerts System

Based on predefined thresholds for vitals (e.g., abnormal heart rate or body temperature), the system can trigger **alerts** in the frontend. These alerts are fetched from the backend and displayed dynamically on the frontend, notifying healthcare providers of critical conditions.

This architecture ensures that the Smart Health Monitoring system works efficiently in monitoring the patient's health remotely, with real-time data visualization and tracking. The integration between the frontend, backend, and sensor simulation ensures that the healthcare providers have immediate access to important health information for timely intervention.

4.2 Data Transmission and Cloud Integration

- Send sensor data wirelessly through http requests or Wi-Fi or Bluetooth modules.
- Store and process the data on a cloud platform (e.g., AWS, Microsoft Azure, MongoDB, or Firebase) for remote access.

4.3 Dashboard Development

- Create a web-based or mobile dashboard to display real-time data for healthcare professionals.
- Dashboard includes:
 - ECG waveform
 - Heart rate graph
 - Temperature graph
 - Body movement indicator
- Implement color-coded indicators (e.g., red for critical, yellow for warning) to highlight abnormal readings.

4.4 Automated Alerts and Notifications

- Integrate a notification system to send instant alerts to medical staff and patients family via SMS, email, or mobile app in emergencies.
- Set predefined threshold values to trigger alerts when vitals deviate from the normal range.

4.5 Predictive Analytics and Machine Learning (Optional)

- Use historical patient data to train machine learning models that can predict health deterioration.
- Implement trend analysis to provide insights and early warnings to medical staff.

4.6 Testing and Validation

- Conduct simulations and real-time tests to validate the accuracy of sensor data and the responsiveness of the alert system.
- Test the system under various scenarios, such as network failure or sensor malfunction, to ensure reliability.

4.7 Security and Data Privacy Implementation

- Apply encryption protocols to protect patient data during transmission and storage.
- Ensure compliance with healthcare standards (like HIPAA or GDPR) to maintain data privacy and confidentiality.

Chapter 5

Requirements

5.1 Hardware Requirements

- **Sensors**
 - 16x2 LCD (I2C)
 - LM35 (Temperature Sensor)
 - MPX10DP (Pressure Sensor)
 - MQ9 MQ135 (Gas Sensors)
 - SPO2 Sensor
 - ECG Sensor
 - Humidity Sensor
 - ADXL (Body Movement Sensor)
 - GPS GSM Modules
 - Buzzer (Alarm)
- **Microcontroller/Development Board**
 - Arduino Mega 2560 or Arduino UNO
- **Communication Modules**
 - Wi-Fi or Bluetooth module (e.g., ESP8266, HC-05)
- **Power Supply**
 - Rechargeable batteries or power adapters for continuous operation
- **Display Unit (Optional)**

- LCD-12(16x2) or OLED screen for onsite data display
- **Cables and Connectors**
 - For wiring sensors to the microcontroller
- **Backup System**
 - Uninterruptible Power Supply (UPS) to avoid downtime in case of power failure

5.2 Software Requirements

- **Programming Language**
 - Python, C/C++, or JavaScript (for coding the microcontroller and dashboards)
- **Cloud Platform**
 - AWS, Microsoft Azure, Firebase, or IBM Cloud for data storage and remote access
- **Database**
 - SQL or NoSQL databases (e.g., MySQL, MongoDB) for storing patient data
- **Dashboard/Interface Development**
 - Web-based tools like HTML, CSS, and JavaScript for UI
 - Mobile app framework (e.g., React Native, Flutter) for real-time monitoring on smartphones

5.3 Network and Communication Requirements

- **Wi-Fi Connectivity**
 - To transmit data from the microcontroller to the cloud
- **Mobile Network (Optional)**
 - For remote monitoring via mobile devices
- **Router or Hotspot**
 - For uninterrupted data transmission

5.4 Security and Privacy Requirements

- **Encryption Algorithms**
 - AES or RSA encryption for secure data transmission
- **Authentication Mechanism**
 - Username-password or biometric login for authorized access to the dashboard
- **Compliance**
 - HIPAA (Health Insurance Portability and Accountability Act) or GDPR compliance to ensure data privacy

Chapter 6

Components and Technologies

The Smart Health Monitoring system integrates various technologies to ensure efficient monitoring of the patient's health and location. Below is an overview of the key components and technologies used in the system:

6.1 Frontend Technologies

The frontend of the Smart Health Monitoring system is responsible for presenting real-time health data and patient location in a user-friendly interface. The following technologies were used to build the frontend:

6.1.1 Frameworks Used

- **React:** React is the core framework used for building the user interface (UI) of the Smart Health Monitoring system. It allows for efficient and dynamic rendering of health data in real time. React components manage the display of various health metrics, patient location, alerts, and historical data. React's declarative approach simplifies the updating of the UI in response to changing data.
- **Bootstrap:** Bootstrap is used to style the UI components, ensuring a responsive and clean design. It provides ready-made components (e.g., buttons, cards, tables) that help in building a professional-looking user interface quickly. Bootstrap's grid system ensures that the layout is responsive across different screen sizes and devices.
- **JavaScript/ES6:** JavaScript and ES6 features (such as arrow functions, promises, and `async/await`) are utilized to manage dynamic content updates and fetch data from the simulation or backend in real time.

6.1.2 Additional Libraries

In addition to the core frontend technologies, the Smart Health Monitoring system also integrates specialized libraries to enhance functionality:

- **Chart.js:** Chart.js is a JavaScript library used to create interactive, animated, and customizable charts for visualizing health metrics in real time. The library is employed to display various graphs (such as ECG waveforms, temperature trends, humidity, and SpO2 levels) to healthcare providers. Chart.js provides a simple and clean interface to render line charts, bar charts, and other visualizations with smooth animations.
- **jsPDF:** jsPDF is used to generate downloadable PDF files containing the patient's medical history. It enables the frontend to capture dynamic data and format it into a printable, shareable PDF document. This is particularly useful for exporting patient records and reports for offline use or sharing with other medical professionals. jsPDF provides functionality to add text, images, and tables to PDFs, making it a versatile tool for document generation.
- **Google Maps API:** The Google Maps API is used to display the patient's real-time location on an interactive map. The frontend fetches the patient's latitude and longitude data and updates the map, allowing healthcare providers to track the patient's location on the go. Markers and info windows are used to show location details like time and vitals on the map.

6.2 Backend Technologies

The backend of the Smart Health Monitoring system is built using Node.js, which handles the processing of incoming requests, management of real-time sensor data, and integration with other services like databases and cloud storage. This section outlines the technologies and structure used in the backend of the system.

6.2.1 Frameworks Used

- **Node.js:** Node.js serves as the runtime environment for the backend, providing a scalable and efficient way to manage asynchronous operations. It is well-suited for building APIs that handle large volumes of data, such as real-time sensor readings in an ICU system.
- **Express.js:** Express.js is used as the framework for routing and handling HTTP requests. It simplifies the process of building RESTful APIs by providing a minimal and flexible approach to route management, middleware integration, and HTTP request handling.

- **Socket.IO:** Socket.IO is utilized to handle real-time communication between the server and the frontend. It enables the backend to emit sensor data to the frontend in real time, providing healthcare providers with live updates on patient vitals.
- **MongoDB:** MongoDB is used as the database to store patient profiles, health records, and real-time sensor data. Its flexibility and scalability make it ideal for managing large amounts of unstructured data, such as sensor logs and medical histories.
- **Cloud Storage (AWS S3/Firebase):** Cloud storage solutions like AWS S3 or Firebase can be used to securely store patient medical data, such as scanned records or diagnostic images, and provide easy access to this data for authorized healthcare providers.

6.2.2 Folder Structure

A well-organized folder structure is essential for maintaining clean, scalable, and manageable code in a backend system. The following is a suggested folder structure for the Node.js backend:

- **controllers/** - Contains route handlers for API requests
- **models/** - Mongoose models for MongoDB
- **routes/** - API route definitions
- **middleware/** - Custom middleware (e.g., authentication)
- **services/** - Business logic and external integrations
- **config/** - Configuration files (e.g., database, environment settings)
- **public/** - Static assets (if any, e.g., uploaded files)
- **app.js** - Main application file (Express setup)
- **server.js** - Server entry point (handles HTTP server setup)
- **package.json** - NPM dependencies and scripts

6.2.3 Middleware

Middleware functions are used to process requests before they reach the final route handler. These are typically used for tasks like authentication, validation, and logging. The following are some examples of middleware that could be implemented:

- **Authentication Middleware:** Checks if a user is authorized to access certain endpoints, using JWT (JSON Web Tokens) for secure token-based authentication.

- **Error Handling Middleware:** Catches errors thrown by route handlers and returns a standardized error response to the client.
- **Request Logging Middleware:** Logs incoming HTTP requests, including method, URL, and timestamps, to help with debugging and monitoring.
- **CORS Middleware:** Handles Cross-Origin Resource Sharing (CORS) to enable the frontend and backend to communicate across different domains or ports.

6.2.4 Additional Libraries Used

In addition to the core Node.js libraries, several additional libraries are used to enhance the functionality of the backend:

- **Mongoose:** Mongoose is an ODM (Object Document Mapper) used for interacting with MongoDB. It simplifies the process of querying and managing MongoDB documents and provides schema-based validation.
- **dotenv:** The dotenv package is used to manage environment variables. It helps store sensitive information like database credentials and API keys securely in a .env file.
- **jsonwebtoken:** This library is used to create and verify JSON Web Tokens (JWT), enabling secure authentication and authorization in the system.
- **axios:** Axios is used to make HTTP requests to external services or other APIs. It's a promise-based HTTP client that simplifies handling requests and responses.
- **nodemon:** Nodemon is used during development to automatically restart the Node.js server whenever code changes are made. This improves development efficiency.
- **bcryptjs:** Bcryptjs is a library used to hash passwords securely, ensuring that user authentication is protected by encryption.

6.2.5 Real-Time Data Streaming with Socket.IO

For the Smart Health Monitoring system, real-time sensor data streaming is implemented using Socket.IO, enabling efficient, bidirectional communication between the server and the frontend. Below is the algorithm for how Socket.IO is used in the system.

How Socket.IO Works

- **Establishing a Connection:**
 - The frontend (React app) establishes a persistent connection to the backend server using Socket.IO.

- The backend listens for a connection event from the client.
- **Emitting Data from the Server:**
 - The server emits sensor data (e.g., temperature, ECG, SpO2) to the frontend at regular intervals or when new data is available.
 - Data is transmitted as JSON objects containing sensor readings.
- **Listening for Data on the Frontend:**
 - The frontend listens for events from the server, specifically for the 'live_sensor_data' event.
 - Upon receiving new data, the frontend updates the state of React components to render the latest sensor data.
- **Handling Disconnections:**
 - If the client or server loses the connection, the frontend listens for the 'disconnect' event.
 - The UI displays a message indicating that the sensor has been disconnected, and attempts to reconnect will be made automatically.
- **Reconnection:**
 - Socket.IO automatically attempts to reconnect if the connection is lost.
 - The frontend listens for the 'connect' event to restore the connection and resume data streaming.

Advantages of Using Socket.IO for Real-Time Data Streaming

- **Low Latency:** Socket.IO allows for real-time, low-latency communication, ensuring that sensor data is delivered instantly to the frontend.
- **Efficient Communication:** It uses WebSockets, reducing the overhead of HTTP requests, leading to faster data updates.
- **Automatic Reconnection:** Socket.IO handles connection interruptions automatically, minimizing data loss or display issues during network issues.
- **Bidirectional Communication:** Enables interactive communication between the client and server, allowing both to send and receive data.

Algorithm Summary

- Step 1: The frontend establishes a connection to the backend server using Socket.IO.
- Step 2: The backend emits real-time sensor data at regular intervals or upon data updates.
- Step 3: The frontend listens for new data and updates the UI with the latest sensor readings.
- Step 4: In case of disconnection, the frontend listens for the 'disconnect' event and notifies the user.
- Step 5: The frontend automatically attempts reconnection using Socket.IO's built-in reconnect functionality.

6.2.6 Containerization with Docker

Docker is a platform for developing, shipping, and running applications inside containers. Containers allow for consistent environments across various stages of development, from local machines to production servers. Docker helps to eliminate issues that arise from differences in environments, ensuring that the application runs the same way no matter where it is deployed.

Docker provides a lightweight and portable method to package an application with its dependencies into a container. This container can then be run on any machine that has Docker installed, making it easier to deploy, scale, and manage applications.

In the Smart Health Monitoring system, Docker is used to containerize the backend services, ensuring that they are portable and can be deployed easily across different environments.

How Docker Works in the System

- **Dockerfile Creation:**
 - A Dockerfile is created to define the environment for the Smart Health Monitoring backend. This file contains instructions on how to build the Docker image.
 - The Dockerfile specifies the base image (e.g., `node:14`) and includes steps such as installing dependencies, copying project files, and setting environment variables.
- **Building Docker Image:**
 - The Docker image is built using the command `docker build -t smart-icu-backend ..`
 - This command processes the Dockerfile, creates a container image, and tags it with the name `smart-icu-backend`.
- **Running the Docker Container:**

- The Docker container is launched using the command `docker run -d -p 3000:3000 smart-icu-backend`.
 - The `-d` flag runs the container in detached mode, and the `-p 3000:3000` flag maps the container's port 3000 to the host's port 3000.
 - The backend is now accessible via `localhost:3000` on the host machine.
- **Docker Compose for Multi-Container Setup:**
 - `docker-compose.yml` is used to manage multiple containers, such as the backend, database (MongoDB), and Redis services.
 - The `docker-compose.yml` file defines the services, networks, and volumes required for the system to run smoothly.
- **Scaling and Deployment:**
 - Docker enables easy scaling of the application by running multiple containers for different services, ensuring fault tolerance and load balancing.
 - Docker images can be pushed to a container registry (e.g., Docker Hub or a private registry) for easy deployment in different environments.

Advantages of Using Docker

- **Portability:** Docker containers are portable and can run consistently across various environments, such as local development, staging, and production.
- **Isolation:** Docker containers provide an isolated environment for each service, reducing conflicts between dependencies.
- **Scalability:** Docker makes it easy to scale services by adding more containers as the demand increases, facilitating load balancing.
- **Faster Development and Deployment:** Docker allows for quick iteration and deployment, as the application environment is consistent across all stages.

Docker Container Image

Algorithm Summary

- Step 1: Create a `Dockerfile` to define the environment for the Smart Health Monitoring backend.
- Step 2: Build the Docker image using the command `docker build -t smart-icu-backend ..`

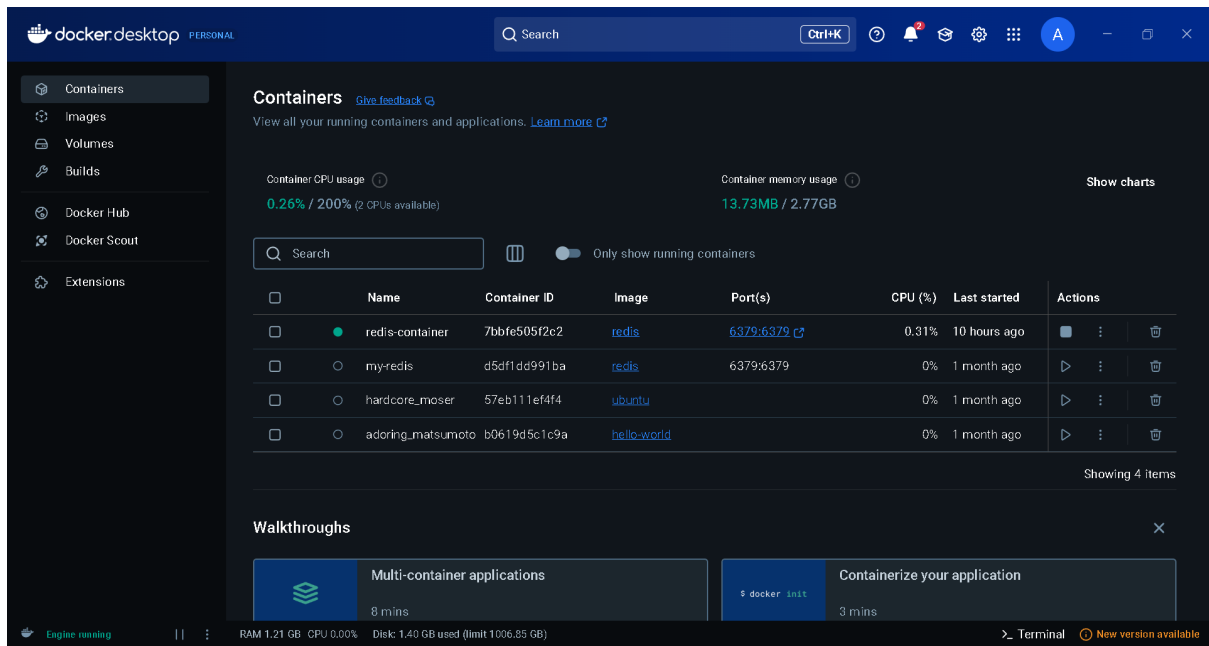


Figure 6.1: Smart Health Monitoring Backend Docker Container

- Step 3: Run the Docker container using `docker run -d -p 3000:3000 smart-icu-backend`.
- Step 4: Use `docker-compose.yml` for managing multiple services such as the backend, database, and Redis.
- Step 5: Scale the application by running multiple containers, ensuring high availability and fault tolerance.
- Step 6: Deploy the Docker container on different environments using Docker Hub or a private container registry.

6.2.7 Redis: In-memory Data Store

Redis (REmote DIctionary Server) is an open-source, in-memory key-value data store. It is commonly used as a cache, message broker, and for storing real-time data. Redis provides high performance, low-latency access to data, which makes it particularly suitable for use cases where fast read and write operations are essential.

Redis is often used in systems where data needs to be accessed quickly, such as in web applications, session storage, real-time analytics, and caching. It supports various data structures such as strings, hashes, lists, sets, and sorted sets, making it flexible for different use cases.

In the context of the Smart Health Monitoring system, Redis has been integrated to handle specific tasks that require fast, real-time data retrieval and processing. Below is an explanation of how Redis is utilized:

What Redis Does

- **Data Caching:** Redis is used to cache frequently accessed data, such as real-time sensor readings and patient health data. By caching this data, Redis reduces the number of requests to the database and ensures that data can be retrieved quickly.
- **Session Management:** Redis can be used for managing user sessions. In the Smart Health Monitoring system, it stores user session information (e.g., authenticated sessions of healthcare providers) to ensure quick access and maintain session state across multiple requests.
- **Real-time Data Streaming:** Redis is employed as a message broker to facilitate real-time data streaming between the backend and the frontend. It helps in broadcasting live updates of sensor readings and health parameters to the UI, ensuring that healthcare providers receive the most up-to-date information without delays.
- **Pub/Sub Model:** Redis implements the Publish/Subscribe (Pub/Sub) messaging pattern, which is useful for systems where one event needs to notify multiple subscribers. In the Smart Health Monitoring, Redis handles events such as sensor threshold breaches (e.g., high temperature or low heart rate) and broadcasts these alerts to the connected frontend clients.

How Redis is Implemented in the Smart Health Monitoring System

- **Redis Installation and Setup:**
 - Redis is installed and configured on the backend server to run as a service. It is then connected to the backend application using the official Redis client for Node.js, such as `redis`.
 - A Redis instance is created and configured to allow connections from the backend.
- **Caching Real-time Sensor Data:**
 - When a new sensor reading is received, the data is stored in Redis for quick access. For example, the sensor reading for temperature might be cached with a key like `temperature:<patientId>`.
 - The data is stored with a TTL (Time To Live) value to automatically expire after a certain period, ensuring that outdated data is not kept in the cache.
- **Pub/Sub for Real-time Alerts:**
 - When a critical health parameter, such as temperature, heart rate, or oxygen levels, exceeds a predefined threshold, the backend system publishes a message to the Redis Pub/Sub channel (e.g., `alert:<patientId>`).

- Frontend clients (e.g., healthcare providers' dashboards) subscribe to this channel to receive alerts in real time.
- The frontend listens for messages and displays an alert whenever critical values are detected.
- **Session Management:**
 - Redis stores user session data such as tokens and authentication states. This ensures that healthcare providers remain authenticated across requests and enables quick session retrieval.

Redis Implementation Algorithm

- Step 1: Install and configure Redis on the backend server.
- Step 2: Establish a connection to Redis using a Redis client in Node.js.
- Step 3: Use Redis to store real-time sensor data, ensuring quick access and retrieval.
- Step 4: Configure Redis to store session information, maintaining user authentication across sessions.
- Step 5: Implement the Pub/Sub model to broadcast real-time alerts to the frontend clients when a health parameter exceeds a threshold.
- Step 6: Set a TTL on cached data to ensure data is refreshed periodically and not kept indefinitely.
- Step 7: Test the system by simulating sensor data to observe Redis caching and Pub/Sub functionality.

Advantages of Using Redis in the Smart Health Monitoring System

- **High Performance:** Redis provides extremely fast data access due to its in-memory storage, ensuring low-latency operations when fetching or updating health data.
- **Scalability:** Redis can handle a large number of simultaneous connections, making it suitable for scaling the Smart Health Monitoring system as the number of connected devices (e.g., sensors and healthcare providers) increases.
- **Real-Time Updates:** With the Pub/Sub model, Redis enables real-time updates of critical health parameters, ensuring that healthcare providers receive timely alerts and information.
- **Data Persistence:** Redis supports persistence options such as snapshotting (RDB) and append-only files (AOF), ensuring that data can be recovered even after a system restart.

Redis Containerization with Docker

Redis can also be containerized using Docker, ensuring that the Redis service is isolated and easily deployable across different environments. A Redis container can be built using the following steps:

- Step 1: Create a Dockerfile for Redis, specifying the base image (e.g., `redis:alpine`).
- Step 2: Build the Docker image using the command `docker build -t redis-container ..`
- Step 3: Run the Redis container using the command `docker run -d -p 6379:6379 redis-container`.
- Step 4: Configure the backend application to connect to the Redis container by specifying the container's IP address or hostname.

6.2.8 MongoDB: Database for Patient Data Storage

MongoDB is a popular NoSQL database designed for storing large amounts of unstructured data in a flexible, scalable manner. Unlike traditional relational databases, MongoDB stores data in JSON-like documents, making it ideal for applications that require high availability, scalability, and flexible data models.

MongoDB is widely used for modern web applications because it can handle large volumes of data with variable structures, allows for easy scaling (both vertical and horizontal), and supports fast read/write operations.

In the Smart Health Monitoring system, MongoDB plays a critical role in storing patient data, historical sensor readings, medical history, and alerts. It acts as the main persistent storage layer, enabling both real-time monitoring and long-term patient record keeping.

What MongoDB Does

- **Stores Patient Profiles:** Each patient's personal and medical information is stored as a document in the database, making it easy to retrieve and update patient details.
- **Stores Real-Time and Historical Sensor Data:** Sensor readings (e.g., heart rate, temperature, oxygen saturation) are logged in the database for both real-time monitoring and future analysis.
- **Stores Medical History:** Long-term health records, including previous treatments, critical incidents, and recovery trends, are stored to assist healthcare providers in decision-making.

- **Stores Alerts and Events:** Critical health alerts are recorded with timestamps, allowing doctors to review the patient's condition history.

How MongoDB is Implemented in the Smart Health Monitoring System

- **Database Connection:**
 - MongoDB is connected to the backend Node.js server using Mongoose, an Object Data Modeling (ODM) library for MongoDB and Node.js.
 - The backend connects to a MongoDB instance (either a local server or a cloud service like MongoDB Atlas) via a connection string.
- **Mongoose Models:**
 - Mongoose schemas and models are defined to represent the structure of data collections such as `Patients`, `SensorReadings`, and `Alerts`.
 - Each model ensures data consistency by validating data types, setting required fields, and establishing relationships between data.
- **CRUD Operations:**
 - The backend APIs perform Create, Read, Update, and Delete operations on the MongoDB database.
 - For example, when a new sensor reading arrives, it is inserted into the `SensorReadings` collection; when a patient's data needs to be updated, a corresponding document in the `Patients` collection is modified.
- **Aggregation and Queries:**
 - MongoDB's powerful aggregation framework is used to calculate patient statistics, generate medical history summaries, and filter critical alerts.

MongoDB Usage Algorithm

- Step 1: Install MongoDB server or use MongoDB Atlas for cloud hosting.
- Step 2: Install Mongoose in the Node.js backend application.
- Step 3: Define Mongoose models for `Patients`, `SensorReadings`, and `Alerts`.
- Step 4: Connect the backend server to the MongoDB instance using a connection string.
- Step 5: Implement API routes to perform CRUD operations on patient data, sensor readings, and alerts.

- Step 6: Use aggregation pipelines to generate reports or calculate summaries when needed.
- Step 7: Ensure error handling and validation during database operations to maintain data integrity.

Advantages of Using MongoDB in the Smart Health Monitoring System

- **Scalability:** MongoDB can handle large volumes of data and can scale horizontally through sharding, which is crucial for managing many patients and high-frequency sensor readings.
- **Flexibility:** The document-based structure allows for easy modifications and additions to the data model without requiring complex schema migrations.
- **High Availability:** MongoDB supports replication, ensuring that patient data remains available even in case of hardware failures.
- **Performance:** Designed for high-performance real-time applications, MongoDB ensures fast read/write operations, which are critical for live health monitoring.

MongoDB Containerization with Docker

MongoDB can be easily containerized using Docker to simplify deployment and management:

- Step 1: Pull the MongoDB Docker image using `docker pull mongo`.
- Step 2: Run the MongoDB container with `docker run -d -p 27017:27017 --name mongo-container mongo`.
- Step 3: Connect the backend application to the MongoDB container using the container's IP address or localhost with the exposed port.

MongoDB Container Image

6.2.9 Arduino IDE: Programming the Sensors

Arduino IDE is an open-source platform used to write, compile, and upload code to Arduino-compatible microcontrollers. It provides a user-friendly environment for developing embedded and IoT applications.

In the Smart Health Monitoring system, Arduino IDE was used to program the Arduino Mega (and optionally ESP8266) to simulate health sensor data and transmit it to the backend server.

Key Functions of Arduino IDE

- **Code Development:** C++ sketches simulate vital parameters like heart rate, SpO₂, and temperature.
- **Data Transmission:** Sends formatted sensor data via USB (Serial) or wirelessly using ESP8266.
- **Backend Integration:** Data is parsed and stored by a Node.js server, then relayed to the frontend in real time.

Implementation Steps

- Install Arduino IDE and board libraries.
- Write sketches using random values for sensor simulation.
- Set up Serial or Wi-Fi communication.
- Format and transmit data at fixed intervals (e.g., 1s).
- Upload the code and verify transmission via Serial Monitor or backend logs.

Advantages

- Beginner-friendly and fast prototyping.
- Supports a wide range of hardware.
- Extensive library and community support.

Chapter 7

Implementation Details

This chapter provides a detailed overview of the implementation of key components in the Smart Health Monitoring system. It covers the React code for different pages, sensor simulation, integration of Google Maps for live location tracking, and the implementation of PDF generation for medical history.

7.1 Code Implementation

Below are some key code snippets that illustrate how various pages in the Smart Health Monitoring system are implemented.

7.1.1 Medical History Page

The medical history page displays the patient's health data over time. Below is the React code to render the medical history in a table format.

Frontend Algorithm: Medical History Display in React

- **Input:** The component receives the medical history data as a prop (i.e., `historyData`).
- **Step 1:** The `MedicalHistory` component receives the `historyData` as a prop.
- **Step 2:** The component renders a table displaying the following columns: Date, Health Parameter, and Value.
- **Step 3:** The `historyData` array is iterated over using the `map` method.
- **Step 4:** For each entry in the `historyData` array:
 - A row is created with three cells: `entry.date`, `entry.parameter`, and `entry.value`.
- **Step 5:** The table is displayed to the user.

- **Output:** The table containing the medical history of the patient is rendered on the screen.

Backend Algorithm: Fetch Medical History Data

- **Input:** The request to fetch the medical history of a specific patient.
- **Step 1:** The backend receives a request for medical history from the frontend.
- **Step 2:** The backend queries the database to retrieve the patient's medical history.
- **Step 3:** The backend processes the database response and organizes the data into a structured format (e.g., date, health parameter, value).
- **Step 4:** The backend sends the structured medical history data as a response to the frontend.
- **Output:** The medical history data is sent back to the frontend, which will display it in the table.

This component takes 'historyData' as a prop, which contains the patient's medical records, and displays them in a table.

7.1.2 Alert System

The alert system monitors critical health parameters such as temperature and heart rate. If any parameter falls outside the normal range, an alert is displayed to notify healthcare providers. Below is the React code for the alert system.

Frontend Algorithm: Alert System in React

- **Input:** The component receives the health data as a prop (i.e., healthData), which includes parameters like temperature and heart rate.
- **Step 1:** The component uses the `useState` hook to initialize the alert state, which stores the current alert message.
- **Step 2:** The `useEffect` hook listens for changes in the `healthData`.
- **Step 3:** When the health data changes, the component checks if:
 - The temperature is above 38°C (high temperature), in which case it sets the alert to `High Temperature! Immediate attention required..`
 - The heart rate is below 60 BPM (low heart rate), in which case it sets the alert to `Low Heart Rate! Immediate attention required..`

- **Step 4:** If no critical condition is detected, the alert is cleared (set to null).
- **Step 5:** If an alert message exists, it is displayed on the screen using a styled alert box.
- **Output:** The alert message is shown to the healthcare provider if a critical health parameter is detected.

Backend Algorithm: Alert System (Critical Health Data Detection)

- **Input:** The backend receives real-time health data from the patient monitoring system (e.g., temperature and heart rate).
- **Step 1:** The backend continuously monitors the health parameters.
- **Step 2:** The backend checks the health parameters to detect any critical conditions:
 - If the temperature exceeds 38°C, the backend considers it as a critical condition (high temperature).
 - If the heart rate is below 60 BPM, the backend considers it as a critical condition (low heart rate).
- **Step 3:** If any of the parameters meet the threshold for a critical condition, the backend generates an alert message, such as `High Temperature! Immediate attention required.` or `Low Heart Rate! Immediate attention required..`
- **Step 4:** The backend then sends this alert message to the frontend via the WebSocket or API for display.
- **Step 5:** If no critical condition is detected, the backend sends a null or clear message to indicate that no alert is needed.
- **Output:** The backend sends an alert message or clears the previous alert based on the health data.

This code triggers alerts based on health data changes, displaying an alert when critical values like high temperature or low heart rate are detected.

7.2 Sensor Simulation

In the absence of physical sensors, we simulate sensor data to mirror the real-world behavior of various health monitoring sensors. These simulated sensors include:

- **Temperature Sensor (LM35):** Random temperature values between 36°C and 38°C simulate the patient's body temperature.

- **ECG Sensor:** Simulated ECG signals that represent the patient's heart rhythm, typically generated using sinusoidal waves or random heart rate values.
- **Heart Rate Sensor:** Simulated heart rate values ranging from 60 bpm to 100 bpm, updated periodically.
- **SpO2 Sensor:** Simulated oxygen saturation levels ranging between 95% and 100%.
- **Other Sensors:** Other health-related metrics like body movement, pressure, and humidity are also simulated using predefined ranges and periodic updates.

These simulated values allow the frontend to reflect real-time monitoring, providing an interactive experience for healthcare providers.

7.3 Live Sensor Readings

The Smart health monitoring system also includes functionality to be able to view the patients live Ecg, Heart rate, blood oxygen, temperature and humidity readings in real time on the website.

Frontend Algorithm: Live Sensor Readings in React

- **Input:** The component receives live sensor data from the backend, which includes values for ECG, temperature, humidity, and SpO2.
- **Step 1:** The component uses the `useState` hook to store the latest sensor readings in states like `ecgData`, `temperatureData`, `humidityData`, and `spo2Data`.
- **Step 2:** The `useEffect` hook is used to fetch initial data from the backend when the component is mounted.
- **Step 3:** The component listens for real-time updates from the backend using `Socket.IO` (via the `socket.on("live_sensor_data")` event).
- **Step 4:** When new sensor data is received from the backend, the component updates the corresponding state variables (e.g., updating `ecgData`, `temperatureData`, etc.).
- **Step 5:** The sensor data is visualized using chart components (such as `Line` for ECG and `Bar` for temperature, humidity, and SpO2), with separate chart options configured for each type of sensor data.
- **Step 6:** If the sensor data is disconnected (i.e., no data is received), a "Sensor Disconnected" message is shown to the user on the respective chart.

- **Output:** The live sensor data is continuously updated and displayed in real-time on the UI, with alerts or warnings for disconnections.

Backend Algorithm: Live Sensor Readings (Using Socket.IO)

- **Input:** The backend receives sensor data from the IoT devices or sensor sources, which includes parameters like ECG, temperature, humidity, and SpO2.
- **Step 1:** The backend sets up a WebSocket server (using Socket.IO) to establish real-time communication with the frontend.
- **Step 2:** The backend continuously listens for new sensor readings from the devices (e.g., through an API, direct sensor input, or a database query).
- **Step 3:** When new sensor data is received, the backend processes and formats the data to ensure it is compatible with the frontend's expectations.
- **Step 4:** The backend emits the sensor data in real-time to the frontend using the Socket.IO `socket.emit("live_sensor_data", data)` method, where `data` contains the latest readings for ECG, temperature, humidity, and SpO2.
- **Step 5:** If the backend detects a disconnection from the sensor devices, it emits a message indicating that the sensor data is unavailable or disconnected.
- **Step 6:** The backend may also periodically check the status of the sensors to ensure that they are still active and transmitting data. If a sensor is disconnected, it triggers an alert to inform the frontend to display a "Sensor Disconnected" message.
- **Output:** The backend sends continuous updates of sensor data to the frontend, ensuring that the UI displays live readings. In case of a sensor failure or disconnection, a disconnection message is sent.

ESP8266 Algorithm

The ESP8266 is a Wi-Fi microchip used to send sensor data to a backend server. Here's the step-by-step algorithm for the ESP8266 to send live sensor readings.

- **Input:** The ESP8266 is connected to a set of sensors (e.g., temperature, humidity, ECG, SpO2).
- **Step 1:** Initialize the sensor libraries on the ESP8266 (e.g., DHT for temperature and humidity sensors).
- **Step 2:** Initialize the Wi-Fi connection using the `WiFi.begin()` function to connect to a network.

- **Step 3:** Create an HTTP or WebSocket client to send data to the backend server.
- **Step 4:** Continuously read data from the sensors using the appropriate functions (e.g., `dht.readTemperature()` for temperature).
- **Step 5:** If data is successfully read, format the sensor data into a JSON object.
- **Step 6:** Send the JSON object containing the sensor data to the backend using an HTTP POST request or WebSocket message.
- **Step 7:** Implement a delay (e.g., `delay(1000)`) to control the frequency of sending data.
- **Step 8:** If the ESP8266 loses the Wi-Fi connection, attempt to reconnect by calling `WiFi.reconnect()`.
- **Output:** The ESP8266 sends real-time sensor data to the backend server.

7.4 PDF Generation

The Smart Health Monitoring system includes the ability to download the patient's medical history as a PDF document. This feature uses the jsPDF library to generate a downloadable PDF. Below is the frontend implementation that demonstrates how this functionality works.

Frontend Implementation

The following code snippet demonstrates how the frontend uses the jsPDF library to generate a PDF file containing the patient's medical history:

PDF Generation Algorithm

- **Input:** Medical history data `historyData`
- **Output:** PDF document `medical_history.pdf`
- **Step 1:** Initialize a new PDF document `doc` using `jsPDF()`.
- **Step 2:** Add the title "Patient Medical History" to the PDF document.
- **Step 3:** For each entry in `historyData`:
 - Retrieve `entry.date`.
 - Retrieve `entry.parameter`.
 - Retrieve `entry.value`.
 - Add `entry.date`, `entry.parameter`, and `entry.value` to the PDF document.

- **Step 4:** Save the document with the file name "medical_history.pdf".
- **Step 5:** Create a button element.
- **Step 6:** On button click, call `GeneratePDF(historyData)`.
- **Step 7:** Trigger the download of the PDF file "medical_history.pdf".

In this code:

- The `downloadPDF` function takes `historyData` (an array of medical history entries) as input.
- A new PDF document is created using the `jsPDF` constructor.
- The title "Patient Medical History" is added at the top of the PDF.
- For each entry in the `historyData`, the date, parameter name, and value are added to the PDF.
- Finally, the `doc.save()` method is used to save the PDF with the filename `medical_history.pdf`.

An example button is provided to trigger the PDF generation when clicked. This button calls the `downloadPDF` function, passing in the `historyData` array.

Backend Considerations

In this implementation, the `jsPDF` library is used purely on the frontend to generate the PDF file. The backend does not participate directly in the PDF generation process, but it plays a critical role in providing the necessary data (i.e., the patient's medical history) to the frontend. Below is a more detailed breakdown of how the backend might handle this task:

- **API Endpoint:** A backend API endpoint (e.g., `GET /api/patient/history`) can be created to allow the frontend to fetch the patient's medical history data. This API might require authentication (e.g., token-based authentication) to ensure that only authorized users can access the sensitive medical data.
- **Data Fetching:** The frontend can make an HTTP request, typically using libraries like `axios` or `fetch`, to retrieve the medical history. The request may include headers for authentication (such as `Authorization: Bearer {token}`) to verify the user's identity and access rights.
- **Return Data:** The backend processes the request and returns the patient's medical history data in a structured format (typically JSON). This data could include various fields such as the patient's name, diagnosis, medical parameters (e.g., temperature, heart rate, blood pressure), medication history, and lab results. The returned data is then used by the frontend to dynamically generate the content of the PDF document.

- **Error Handling:** The backend should also include proper error handling mechanisms, ensuring that if something goes wrong during data retrieval (e.g., missing data or server errors), the frontend can handle such issues gracefully by displaying an appropriate error message or retrying the request.
- **Security Considerations:** Since medical data is sensitive, the backend must ensure that the data is stored securely and that data transmissions between the frontend and backend are encrypted, typically using HTTPS. Additionally, the backend should implement role-based access control (RBAC) to limit access to the patient's medical history based on the user's role (e.g., doctor, nurse, patient).

7.5 Redis Implementation and Server Load Balancing

Redis has been successfully integrated into the backend of the Smart Health Monitoring system to optimize performance, manage real-time sensor data efficiently, and reduce server load. Redis is an in-memory key-value store that provides extremely fast read and write operations, making it ideal for caching and handling high-frequency data.

7.5.1 How Redis is Implemented

- A Redis server was deployed using Docker to ensure a consistent, isolated environment.
- The Node.js backend connects to the Redis server using the `redis` NPM library.
- Real-time sensor data (e.g., heart rate, temperature, oxygen saturation) is cached in Redis instead of querying MongoDB directly for every request.
- Redis keys are set with appropriate expiration times (TTL) to ensure that only fresh, up-to-date data is stored and served.
- In case of new sensor data arrivals, the backend updates the cache in Redis immediately, ensuring that the frontend receives near-instantaneous updates.

7.5.2 How Redis Helps the Smart Health Monitoring System

- **Load Reduction on MongoDB:** By serving the most recent data directly from Redis, the number of read requests hitting the MongoDB database is significantly reduced, leading to faster response times and a lower overall server load.
- **Faster Real-Time Data Access:** Redis enables extremely fast retrieval of sensor data for frontend display, making the system feel more responsive and suitable for critical monitoring in ICU environments.

- **Efficient Data Updates:** Real-time updates from Arduino/ESP8266 simulations are quickly written to Redis, ensuring that healthcare providers see the latest readings without delay.
- **Scalability:** With Redis acting as a caching layer, the backend can easily handle a higher number of simultaneous frontend users without performance bottlenecks.

7.6 Docker Implementation and Containerization

Docker has been successfully integrated into the Smart Health Monitoring system to ensure consistent, portable, and isolated deployment environments across different systems. Docker simplifies the management of services like Redis and the Node.js backend, making the overall infrastructure more reliable and easier to maintain.

7.6.1 How Docker is Implemented

- Docker containers were created for services such as the Node.js backend server and Redis server.
- Docker images were built based on official images like `node:latest` and `redis:latest`.
- A `Dockerfile` was used to define the environment, dependencies, and startup commands for the backend application.
- `docker-compose` was utilized to orchestrate multiple services (backend, Redis) together, making it easy to launch or shut down the system with a single command.
- Port mappings and environment variables were configured to enable smooth communication between services inside and outside containers.

7.6.2 How Docker Helps the Smart Health Monitoring System

- **Consistency Across Environments:** Docker ensures that the Smart Health Monitoring system runs the same way on different machines without the "it works on my machine" problem.
- **Simplified Deployment:** Deployment of the backend and Redis servers becomes faster and more reliable using containerized services.
- **Isolation:** Each service runs in its own isolated environment, reducing conflicts between dependencies and improving security.
- **Scalability and Maintenance:** New services (e.g., AI modules, load balancers) can easily be added in the future by updating the Docker Compose configuration.

7.7 MongoDB Implementation and Data Management

MongoDB has been integrated into the Smart Health Monitoring system as the primary database for storing structured and semi-structured patient health data. MongoDB is a NoSQL, document-oriented database that provides flexibility, scalability, and high performance, making it ideal for storing dynamic health monitoring data.

7.7.1 How MongoDB is Implemented

- A MongoDB server was deployed, either locally or using a cloud-hosted solution such as MongoDB Atlas.
- The Node.js backend connects to MongoDB using the mongoose library, which provides an elegant object modeling tool.
- Dedicated collections were created for different types of data, such as patient profiles, sensor readings, and medical histories.
- Schemas were designed using Mongoose models to define the structure and validation rules for data being stored.
- CRUD (Create, Read, Update, Delete) operations were implemented to handle real-time updates and historical queries of patient data.

7.7.2 How MongoDB Helps the Smart Health Monitoring System

- **Flexible Data Storage:** MongoDB's document model allows storage of varying health parameters without rigid schema constraints.
- **Real-Time Data Handling:** The database efficiently handles frequent insertions and updates from live sensor feeds.
- **Scalability:** MongoDB can scale horizontally across multiple servers, making it suitable for future expansions involving larger numbers of patients and sensors.
- **Easy Data Querying:** MongoDB's powerful query language allows healthcare providers to quickly retrieve historical trends, generate reports, and conduct data analysis.

7.8 Arduino Code Implementation and Real Sensor Data Acquisition

Arduino boards were used in the Smart Health Monitoring system to acquire real-time patient sensor data such as heart rate, body temperature, and oxygen saturation. Programming was

done using the Arduino IDE with C++-based sketches.

7.8.1 How Arduino Code is Implemented

- The Arduino Mega board was programmed using the Arduino IDE to read live data from connected biomedical sensors.
- Sensors such as pulse sensors, temperature sensors (e.g., LM35/DS18B20), and SpO2 sensors were interfaced with the Arduino's analog and digital pins.
- Raw sensor readings were processed if necessary (e.g., conversion of analog voltage to temperature).
- The collected sensor data was structured into a consistent format (e.g., JSON or comma-separated strings) for transmission.
- Data was sent over the Serial port at regular intervals to the backend server or to the ESP8266 module for wireless transmission.

7.8.2 How Arduino Code Helps the Smart Health Monitoring System

- **Real Data Acquisition:** Provides authentic, real-world sensor readings crucial for patient monitoring and system validation.
- **Continuous Monitoring:** Enables ongoing data capture essential for real-time health tracking and alert generation.
- **Data Formatting and Preprocessing:** Ensures that the sensor output is clean and structured for efficient backend processing.
- **Integration with Wireless Modules:** Easily connects with modules like ESP8266 for remote data transmission.

7.9 ESP8266 Code Implementation and Wireless Transmission of Real Sensor Data

The ESP8266 Wi-Fi module was integrated into the Smart Health Monitoring system to enable wireless transmission of real patient sensor data, improving the mobility and flexibility of the system.

7.9.1 How ESP8266 Code is Implemented

- The ESP8266 was programmed using the Arduino IDE with the appropriate ESP8266 board packages.
- The module establishes a connection to a secure Wi-Fi network using preconfigured SSID and password credentials.
- Real-time sensor readings from the Arduino (received via Serial or direct sensor connection) were transmitted wirelessly to the backend server.
- The sensor data was packaged in JSON format to maintain consistency and ease of parsing.
- WebSocket or HTTP protocols were used for efficient and reliable real-time communication with the server.
- Automatic reconnection logic was implemented to maintain stability in case of Wi-Fi disruptions.

7.9.2 How ESP8266 Code Helps the Smart Health Monitoring System

- **Wireless Communication:** Allows sensor data to be transmitted without physical wiring, increasing system flexibility.
- **Real-Time Data Streaming:** Sends authentic sensor readings live to the backend for immediate visualization and analysis.
- **Enhanced Scalability:** Supports the deployment of multiple wireless units for monitoring several patients across an ICU facility.
- **Robust Connectivity:** Maintains consistent communication even in variable network environments, ensuring no data loss.

Chapter 8

Results and Discussion

8.1 Screenshots/Visuals

In this section, we provide screenshots of the key components of the system, including the patient-side user interface (UI), the dashboard, the GPS tracking page, the alert system, and the medical history page. These visuals demonstrate the system's ability to track real-time patient data and display it in a user-friendly interface. The following screenshots capture these functionalities:

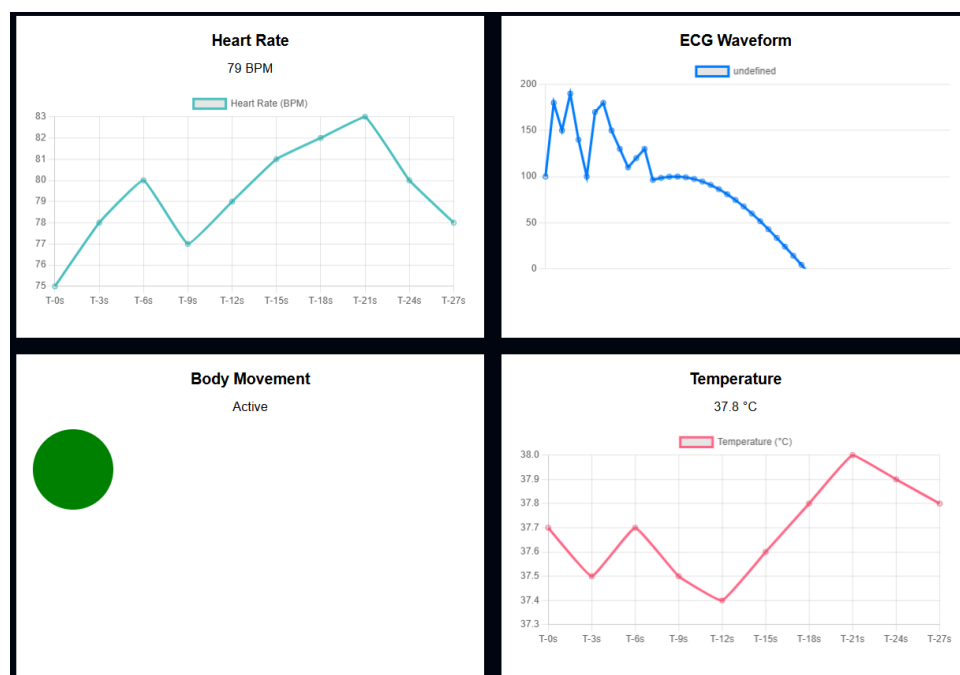


Figure 8.1: Patient-Side Dashboard displaying Health Metrics and Patient Status

8.2 Testing and Validation

The system underwent thorough testing to ensure its functionality and accuracy in real-world scenarios. Below are the major areas of testing and validation:

8.2.1 Patient Location Tracking

The GPS tracking feature was extensively tested by simulating different movements for the patient's location. We observed that the system accurately displayed the patient's location on the Google Map, with updates occurring at regular intervals. The simulated movements (latitude and longitude updates) were tested in different regions to ensure that the real-time tracking was functional and responsive.

Location tracking is a critical aspect of the Smart Health Monitoring system, especially for patients who may be mobile or at risk of wandering. Real-time tracking of a patient's location provides healthcare providers with valuable data on the patient's whereabouts at all times, enhancing patient safety. For example, if a patient with critical health conditions moves unexpectedly or leaves a designated area, the system can alert medical staff and provide immediate assistance.

Moreover, location tracking can be particularly useful in emergency situations. In cases where a patient's condition deteriorates rapidly, knowing their exact location allows for quicker intervention, potentially saving lives. It also helps in managing patients in large healthcare facilities or even remotely monitoring patients at home, ensuring that healthcare professionals can always provide timely support and care.

By using GPS for real-time tracking, the system can improve overall patient care, facilitate quicker responses in emergencies, and ensure that healthcare providers are always aware of the patient's location, especially in critical or unpredictable situations.

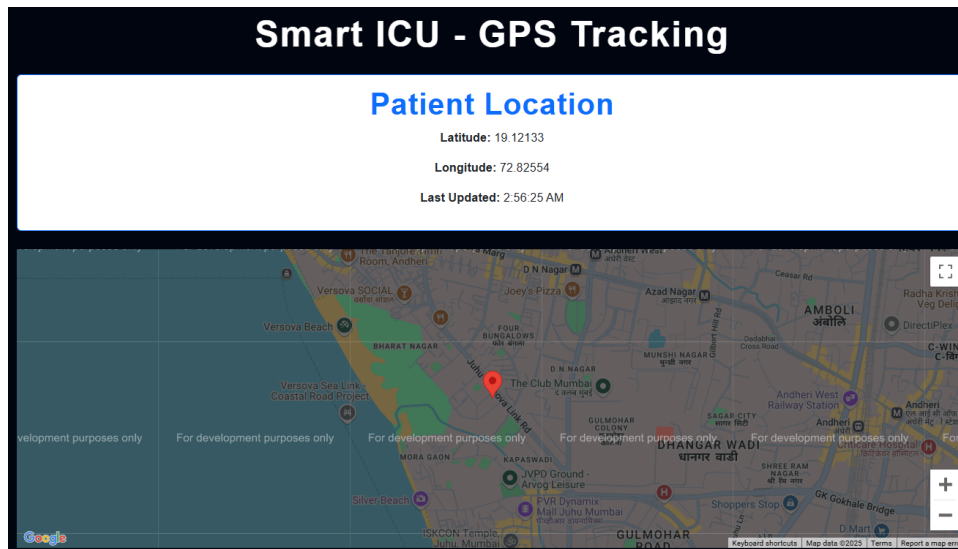


Figure 8.2: GPS Tracking Page showing Patient's Live Location on Google Maps

8.2.2 Medical History Management

We tested the medical history management feature by simulating the entry and display of patient health data, such as heart rate, body temperature, and oxygen saturation (SpO2) levels. The medical history page effectively stored and presented these data points, showing a continuous record of the patient's health over time. Additionally, the option to download the medical history as a PDF was verified to ensure the functionality was seamless.

Medical history management is a crucial component of any healthcare system, especially in the context of continuous patient monitoring. By tracking a patient's health data over time, healthcare providers gain insights into trends, helping them to better understand the patient's condition and make more informed decisions. For example, historical data on body temperature, heart rate, and oxygen levels can indicate whether a patient's condition is improving or deteriorating, which is vital for timely interventions.

Additionally, having a centralized system to store and view patient data allows for better coordination among healthcare professionals. This is especially important in situations where multiple caregivers are involved, as it ensures that everyone has access to the most up-to-date information. Furthermore, the ability to download the medical history as a PDF provides an easy way to share the patient's records with other medical professionals or keep them for personal reference.

By providing both real-time data and historical context, the medical history feature ensures that healthcare providers can make decisions based not only on current health readings but also on past trends, contributing to more personalized and effective care.

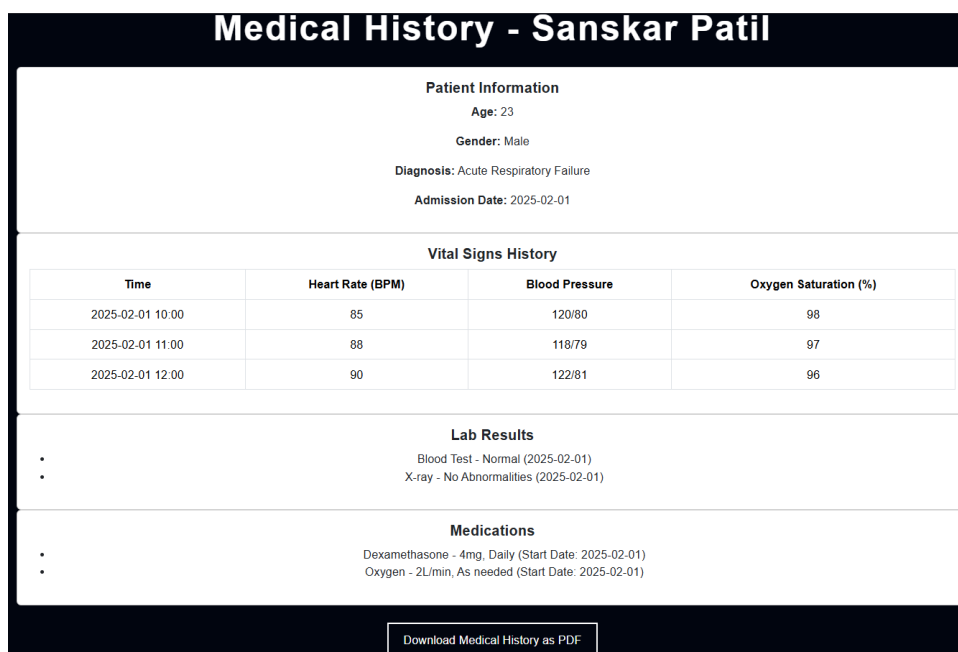


Figure 8.3: Medical History Page displaying Historical Health Data and Patient’s Health Profile

8.2.3 Alert System

The alert system was tested by simulating abnormal sensor readings. For instance, we simulated high body temperature and abnormal heart rate values, and the system successfully triggered alerts accordingly. These alerts were displayed on the frontend, notifying healthcare providers about the critical condition of the patient.

The alert system plays a crucial role in ensuring timely medical interventions, particularly in cases where a patient’s condition deteriorates rapidly. By continuously monitoring vital signs such as heart rate, oxygen levels, and body temperature, the system can detect anomalies and immediately notify caregivers. This feature is especially important for patients with chronic illnesses, elderly individuals, or those in critical care units.

In real-world scenarios, delays in responding to critical health changes can lead to severe complications or even fatalities. The alert mechanism minimizes this risk by providing instant notifications through visual and auditory signals. The system is designed to trigger alerts based on predefined threshold values. For example, if the heart rate drops below or exceeds a safe range, or if oxygen saturation falls below critical levels, an immediate warning is issued.

Furthermore, integrating the alert system with a digital healthcare platform ensures that notifications can be sent not only via the patient-side dashboard but also through mobile notifications, emails, or direct alerts to medical personnel. This guarantees that healthcare providers can respond promptly, even if they are not physically present with the patient.

By automating the detection of abnormal health conditions and facilitating real-time responses, the alert system significantly enhances patient safety and reduces the burden on healthcare staff.

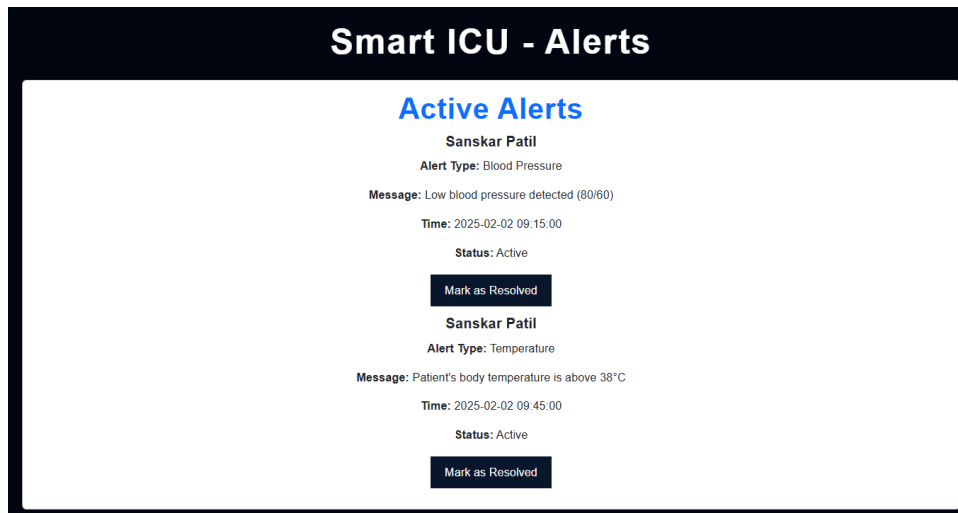


Figure 8.4: Alert System Page showing Critical Health Alerts such as Abnormal Temperature or Heart Rate

8.3 Challenges and Solutions

Throughout the development process, we faced a few challenges, which were resolved as follows:

8.3.1 Google Maps Integration

One of the main challenges was integrating Google Maps into the frontend to display the patient's live location. Initially, we faced issues related to API key configuration and display responsiveness. This was resolved by ensuring the correct API key was used and setting proper parameters for rendering the map. The use of '@react-google-maps/api' helped streamline the process and provide easy-to-implement mapping features.

8.3.2 Managing Simulation Data

Since we simulated sensor data for the prototype, managing dynamic and realistic data for multiple sensors such as temperature, heart rate, SpO2, and others posed a challenge. We overcame this by writing JavaScript code that simulated sensor readings and updated them at regular intervals, mimicking real-world data flow.

8.3.3 Managing Cached Live Sensor Data

While working with real-time sensor data from connected biomedical sensors, we encountered the challenge of efficiently storing and retrieving the most recent sensor readings without overloading the database or increasing server response time.

Since sensor data updates frequently and needs to be instantly available for the frontend

display and alert system, directly querying the MongoDB database for every sensor update would have led to high latency and performance bottlenecks.

To address this, we implemented a caching mechanism using Redis:

- **Need for Cached Memory:** Live sensor readings required temporary, fast-access storage so that the backend could quickly serve the latest data without frequent database reads.
- **Problem with Direct Database Access:** Querying MongoDB for every frontend request would have caused unnecessary load, increased response time, and could lead to database contention as the system scaled.
- **Solution with Redis:** Redis, being an in-memory key-value store, was used to cache the latest sensor readings for each patient. Sensor data updates were written immediately into Redis, and the frontend fetched data directly from Redis for real-time display.
- **Efficiency Gains:** This reduced the number of MongoDB queries dramatically, improved system responsiveness, and allowed the backend to handle a larger number of simultaneous user connections.

By strategically caching only the latest data with appropriate expiration times (TTL), we ensured both data freshness and system scalability, making Redis a crucial part of the real-time Smart Health Monitoring architecture.

8.3.4 Managing Server Load with Real-Time Streaming

During the development of the Smart Health Monitoring system, another significant challenge was handling continuous frontend API requests for live sensor data updates. If the frontend repeatedly polled the server for new data, it would lead to:

- **Increased Server Load:** Frequent HTTP requests from multiple clients could overwhelm the server, causing performance degradation or even crashes.
- **Higher Latency:** Polling introduces delays between data generation and data retrieval, making the system less responsive for critical ICU monitoring.
- **Resource Inefficiency:** Handling numerous unnecessary API calls would waste bandwidth and CPU resources on both client and server sides.

To solve this, we integrated **Socket.IO** for real-time, event-driven communication between the backend and frontend:

- **Persistent Connection:** A WebSocket connection was established between the server and each connected client, enabling continuous two-way communication without repeated HTTP requests.

- **Real-Time Data Push:** Instead of the frontend pulling data, the backend pushes new sensor readings to all connected clients immediately when new data is available.
- **Reduced Server Strain:** By eliminating the need for constant API polling, server load decreased significantly, improving system scalability and responsiveness.
- **Instantaneous Updates:** Healthcare providers receive sensor data and alerts with minimal delay, enhancing the reliability of the Smart Health Monitoring system.

The integration of Socket.IO ensured a stable, efficient, and truly real-time system suitable for critical ICU environments where every millisecond matters.

8.3.5 Efficient Storage of Live Sensor Data

Another key challenge faced during the development of the Smart Health Monitoring system was managing the storage of continuously incoming live sensor data. Attempting to store every real-time reading in a persistent database like MongoDB would lead to:

- **High Storage Costs:** Continuously inserting real-time data would rapidly consume database storage, leading to increased operational costs, especially for cloud-hosted solutions.
- **Database Performance Issues:** Massive volumes of small, frequent write operations could degrade database performance over time, affecting query speed and system reliability.
- **Unnecessary Data Retention:** In many cases, only the latest sensor readings are critical for real-time monitoring, and historical data beyond a short time window is not immediately needed.

To address this, we implemented a caching strategy using **Redis**:

- **Short-Term Data Storage:** Instead of storing every incoming reading permanently, we cached only the latest 1-minute worth of sensor data in Redis.
- **Expiration Policy:** Each cached entry was set with a Time-To-Live (TTL) of 60 seconds. After this period, old sensor data would automatically expire and be deleted from the cache.
- **Optimized Database Usage:** Critical historical trends or significant alerts are selectively stored in MongoDB, while routine sensor readings are handled by Redis caching.
- **Real-Time Frontend Access:** The frontend retrieves the latest available data directly from Redis, ensuring instant updates without burdening the main database.

This solution significantly reduced storage requirements, improved system performance, and ensured that the Smart Health Monitoring system could handle real-time sensor feeds efficiently without excessive resource consumption.

8.3.6 User Interface and Responsiveness

Ensuring that the patient-side user interface (UI) was both intuitive and responsive across different devices was another challenge. To address this, we utilized Bootstrap for responsive design and ensured that the UI elements were correctly aligned and adjusted according to screen size. This allowed us to maintain a clean and functional UI on both desktop and mobile devices.

Chapter 9

Future Work and Enhancements

9.1 Potential Improvements

While the current system effectively simulates a smart patient monitoring environment, several enhancements can be made to improve its accuracy, functionality, and real-world applicability. Some of the key areas for improvement include:

9.1.1 Integration of Real Sensors

Currently, the system relies on simulated sensor data to mimic real-world conditions. A significant enhancement would be replacing simulated data with real sensor inputs. By integrating actual medical sensors such as:

- Real-time ECG monitors (e.g., AD8232) for precise heart rate tracking.
- Pulse oximeters (e.g., MAX30100) for accurate SpO2 measurements.
- Blood pressure sensors to provide continuous BP monitoring.
- Wearable accelerometers (e.g., MPU6050) for detecting patient movement and fall detection.

This would allow for more reliable data collection and better accuracy in patient health monitoring.

9.1.2 Advanced Alert System with AI Integration

The current alert system is based on threshold values for different health parameters. Future enhancements can incorporate Artificial Intelligence (AI) and Machine Learning (ML) to:

- Detect patterns in a patient's health data over time.
- Predict potential health risks based on historical trends.

- Reduce false alarms by distinguishing between critical conditions and temporary fluctuations.

AI-driven alerts would make the system more intelligent and capable of providing early warnings for potential health risks.

9.1.3 Cloud-Based Data Storage and Remote Access

At present, patient data is managed on the frontend. A cloud-based infrastructure can be implemented to:

- Store real-time patient data securely on cloud servers.
- Enable healthcare providers to access patient history from any location.
- Allow for data analysis and trend visualization over extended periods.

Integration with platforms like Firebase, AWS IoT, or Google Cloud would enhance scalability and data accessibility.

9.1.4 Mobile App Integration

To improve accessibility, a mobile application can be developed alongside the web interface. This would allow:

- Patients and caregivers to receive real-time alerts and notifications.
- Doctors to monitor patient vitals remotely from their smartphones.
- Location tracking via GPS to be more seamless and efficient.

9.1.5 Scalability for Large-Scale Implementation

For real-world deployment in hospitals or remote healthcare facilities, the system should be scalable to support:

- Multiple patients being monitored simultaneously.
- Secure multi-user authentication for different levels of access (e.g., doctors, nurses, family members).
- Integration with hospital databases and Electronic Health Records (EHRs).

Ensuring scalability would make the system applicable in both home-based healthcare and hospital environments.

9.1.6 IoT and Wearable Device Connectivity

Future enhancements could involve connecting the system with wearable healthcare devices like:

- Smartwatches with heart rate and SpO2 monitoring.
- IoT-enabled glucose monitors for diabetic patients.
- Bluetooth-enabled blood pressure cuffs.

These integrations would provide continuous health tracking without requiring the patient to be physically wired to monitoring devices.

Chapter 10

Conclusion

The Smart Health Monitoring System overcomes the limitations of traditional critical care by offering continuous, real-time, and patient-specific monitoring. By integrating IoT sensors, cloud computing, and mobile communications, the system enables instant alerts and remote access, allowing physicians to respond swiftly to emergencies and identify potential health risks before they escalate.

The scalability of this design ensures it can be adapted to different hospital environments, making it a cost-effective and efficient solution. By reducing the burden on medical staff and enhancing patient safety, the Smart Health Monitoring system improves healthcare outcomes.

Our advanced health monitoring system incorporates multiple sensors, including LM35 temperature sensors, MEMS accelerometers, SPO2 sensors, ECG modules, gas sensors, and motion detectors. These components facilitate the real-time tracking of vital health indicators such as temperature, oxygen levels, heart activity, air quality, and movement, ensuring proactive and timely medical intervention.

Additionally, this solution provides healthcare professionals with secure, direct access to critical patient data, enabling them to make informed decisions swiftly. Despite challenges such as data security and integration complexities, our Smart Health Monitoring system enhances patient care, optimizes medical workflows, and elevates healthcare standards, ultimately leading to better patient outcomes and hospital efficiency.

References

1. E. Al Alkeem et al., "New secure healthcare system using cloud of things," Springer Science+Business Media New York, 2017.
Available at: https://www.researchgate.net/publication/316749947_New_secure_healthcare_system_using_cloud_of_things
2. Y. Kim, S. Lee, and S. Lee, "Coexistence of ZigBee-based WBAN and WiFi for Health Telemonitoring Systems," IEEE Journal of Biomedical and Health Informatics, DOI: 10.1109/JBHI.2014.2387867.
3. M. M. Baig and H. Gholamhosseini, "Smart Health Monitoring Systems: An Overview of Design and Modeling," Springer Science+Business Media New York, 2013.
4. S. M. Riazulislam et al., "The Internet of Things for Health Care: A Comprehensive Survey," IEEE Transactions, DOI: 10.1109/TDSC.2015.2406699.
Available at: <https://www.semanticscholar.org/paper/The-Internet-of-Things-for-Health-3AA-Survey-Islam-Kwak/cddb22908f28a1636cbbdeb3a4f0e00f9cef05a9>
5. A. Mdhaftar et al., "IoT-based Health Monitoring via LoRaWAN," IEEE EUROCON 2017.
Available at: https://www.researchgate.net/publication/319169748_IoT-based_health_monitoring_via_LoRaWAN
6. M. M. Masud et al., "Resource-Aware Mobile-Based Health Monitoring," 2015 IEEE.
7. A. Bansal et al., "Remote health monitoring system for detecting cardiac disorders," IET Syst.Biol., vol. 9, no. 6, pp. 309–314, 2015.
8. H. Al-Hamadi and I. Chen, "Trust-Based Decision Making for Health IoT Systems," IEEE Internet of Things Journal, DOI: 10.1109/JIOT.2017.2736446.

Originality Report



Page 2 of 59 - Integrity Overview

Submission ID trn:oid::18152:93349464





5% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.




Filtered from the Report

- ▶ Bibliography
- ▶ Quoted Text
- ▶ Cited Text
- ▶ Small Matches (less than 12 words)

Match Groups

-  **39 Not Cited or Quoted 5%**
Matches with neither in-text citation nor quotation marks
-  **0 Missing Quotations 0%**
Matches that are still very similar to source material
-  **0 Missing Citation 0%**
Matches that have quotation marks, but no in-text citation
-  **0 Cited and Quoted 0%**
Matches with in-text citation present, but no quotation marks

Top Sources

- 2%  Internet sources
- 0%  Publications
- 5%  Submitted works (Student Papers)

Integrity Flags

0 Integrity Flags for Review

No suspicious text manipulations found.

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

Bibliography

- [1] Baig, M. M., Gholamhosseini, H. (2013). Smart health monitoring systems: an overview of design and modeling. **Journal of medical systems**, 37(2), 9898.
- [2] Kim, J., Park, H., Yoo, S. (2014). Coexistence of ZigBee-based WBAN and WiFi for health telemonitoring systems. **IEEE Transactions on Biomedical Circuits and Systems**, 8(4), 565–573.
- [3] Riazulislam, M., Kwak, D., Humaun Kabir, M., Hossain, M., Kwak, K. S. (2015). The Internet of Things for health care: A comprehensive survey. **IEEE Access**, 3, 678–708.
- [4] Al Alkeem, E., Al Falasi, H., Al Zaabi, M., Al Shamsi, A. (2017). A secure healthcare system for patient data based on cloud of things. **Procedia Computer Science**, 110, 206–213.